(4)

AD-A213 842

Technical Report 1310
August 1989

Periodic Phase Adjustment Distributed Clock Synchronization in the Hard Realtime Environment

D. R. Wilcox

209

ADMINISTRATIVE INFORMATION

This report was carried out by the Embedded Computer Systems Branch, Code 412, of the Naval Ocean Systems Center, San Diego, CA, under program element number 0602234N.

Released by
L. J. Core, Head
Embedded Computer Systems
Branch

Under authority of
A. G. Justice, Head
Information Processing and
Displaying Division

ACKNOWLEDGMENT

The author wishes to thank the members of the IEEE 896 Futurebus+ Committee, and especially Dr. Lui Sha of the Advanced Real-Time Technology (ART) Project of the Software Engineering Institute at Carnegie Mellon University and Dr. Richard Volz of the Department of Computer Science at Texas A & M University for their technical guidance and support. The ART project is the primary source of development for the rate-monotonic hard-deadline scheduling technology summarized in Section 3.

FS

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED		1b. RESTRICTIVE MARKINGS	
2a. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.	
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE			
4. PERFORMING ORGANIZATION REPORT NUMBER(S) NOSC TR 1310		5. MONITORING ORGANIZATION REPORT NUMBER(S)	
6a. NAME OF PERFORMING ORGANIZATION Naval Ocean Systems Center	6b. OFFICE SYMBOL (if applicable) Code 412	7a. NAME OF MONITORING ORGANIZATION	
6c. ADDRESS (City, State and ZIP Code) San Diego, California 92152-5000		7b. ADDRESS (City, State and ZIP Code)	
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Naval Ocean Systems Center	8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS (City, State and ZIP Code) San Diego, California 92152-5000		10. SOURCE OF FUNDING NUMBERS	

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	AGENCY ACCESSION NO.
0602234N	ECB1		DN306 243

11. TITLE (include Security Classification)

PERIODIC PHASE ADJUSTMENT DISTRIBUTED CLOCK SYNCHRONIZATION IN THE HARD REALTIME ENVIRONMENT

12. PERSONAL AUTHOR(S)

D. R. Wilcox

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM TO	August 1989	27

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	distributed processing system simultaneous clock sampling

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report describes the operation and rationale underlying an approach to synchronizing a set of time-of-day realtime clocks located on the respective processors of a distributed processing system. The approach is called periodic phase adjustment. The report also shows how to integrate this approach into rate monotonic hard-deadline realtime scheduling technology.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL D. R. Wilcox	22b. TELEPHONE (include Area Code) (619) 553-5467	22c. OFFICE SYMBOL Code 412

TABLE OF CONTENTS

TABLE OF FIGURES

1.0 INTRODUCTION

Many multiprocessing and distributed-processing system applications require the various components of the system to have access to the same knowledge of the current time of day. Uses for time-of-day data include the generation of timestamps for the coordination of transactions in a database system, the detection of process faults without the limitations of timeouts in distributed fault-tolerant systems [Lamport, 1984], and the recording and predicting of physical world events in a realtime system. There are various alternative methods of coordinating database transactions which avoid time-of-day timestamps, such as traditional locking mechanisms and timestamps employing virtual time [Jefferson, 1985]. There is no alternative for time-of-day facilities in a system designed to act upon real-world, time-oriented events.

This paper describes the operation and rationale underlying an approach to synchronizing a set of time-of-day realtime clocks located on the respective processors of a distributed processing system. The approach is called **periodic phase adjustment** [1]. The paper also shows how to integrate this approach into rate monotonic hard-deadline realtime scheduling technology.

1.1 CENTRALIZED VERSUS DISTRIBUTED CLOCK

A system with a centralized clock which requires less hardware overall may seem better than replicating the clock locally at each module. A centralized clock also avoids the problem of synchronizing the collection of local clocks. However, there are three reasons why a distributed set of local clocks may be a better approach.

First, access by a processor to a centralized clock is generally slower than access to a local clock connected more directly to the processor. The degradation in speed is due to the physically longer access path to the centralized clock and the many potential software and hardware interfaces between the processor and the centralized clock.

Second, time values obtainable from a centralized clock are usually less accurate than those obtainable from a local clock. The centralized clock, by definition, is a shared resource. Contention may arise between users of the centralized clock or some component of the access path to the centralized clock. Since competing traffic may be neither interruptible nor of known duration, it is often impossible to determine when the request for time data actually reached the centralized clock. This calls into question the accuracy of the time data obtained.

The objection may be raised that systems requiring accurate time data employ a centralized time standard, such as a cesium clock, which is considerably more accurate than a collection of local clocks driven by relatively low-cost crystal oscillators. The issue of contention at the centralized clock, or on the access path to it, degrading the obtainable accuracy (regardless of the internal accuracy of the centralized clock itself) is still valid. These systems can employ the external time standard for accuracy, yet still gain the benefits of distributed local clocks, provided that they implement tight synchronization between the local clocks and the external time standard.

[1] The "phase" concept originated from an analogy of the analog "phase lock loop" circuit for the clock synchronization approach described below.

Third, a centralized clock is less fault tolerant than a collection of local clocks. The centralized clock represents a single point of failure. A distributed collection of local clocks not only provides a backup source of time data should the primary source fail, but also provides a means of validating time data by comparing clocks.

Distributed local clocks avoid the three problems described above, but introduce a new problem, the need to keep the local clocks synchronized with one another or with an external time standard. The remaining sections of this paper assume the distributed clock approach.

2.0 LOCAL CLOCK ADJUSTMENT

The synchronization of a system's clocks, whether it be with each other or with an external time standard, implies that the clocks be adjustable. This section concentrates on the local clock itself and describes the nature of local clock adjustment and how it can be implemented.

2.1 CLOCK VALUE VERSUS CLOCK RATE

There are two methods for adjusting a clock in order to bring it into synchronization. These methods can be illustrated by analogy to adjusting a mechanical clock powered by a spring. The first method is to move the hands on the clock face. This is equivalent to modifying the *value* displayed by the clock. The second method is to adjust the tension on the spring powering the clock (usually by a screw adjustment on the back). This is equivalent to modifying the *rate* at which the time displayed advances. One can also use a combination of the two methods [2].

Once a clock value has been initialized, adjusting the clock rate is usually superior to adjusting clock value for several reasons.

First, adjusting clock rate avoids backward adjustments in time value. When the clock runs too fast, it displays a time beyond the time desired. Adjusting the clock value to the proper time implies moving the time value backward for an instant. This destroys the utility of the clock in applications that depend on timestamps to order a sequence of events or transactions. Adjusting clock rate, on the other hand, never causes the clock value to move backward. When the clock has been running too fast, the rate is adjusted so that the value advances more slowly. The time value gradually converges toward the proper value. As the time value approaches the proper value, future rate adjustments converge on the proper rate to maintain the proper value.

Second, adjusting clock rate avoids abrupt discontinuities in the otherwise steady advance of clock value. The problem with abrupt discontinuities can be illustrated by a simple example. Consider computing the average speed of an object moving in a straight line between two points. The average speed is the distance between the two points divided by the difference in the time values sampled when the object was at each point. If the clock value is dramatically adjusted just after the object leaves the first point but just before the object arrives at the second point, the computed speed could be in error. Even worse, a backward adjustment in time value could cause the computed speed to be negative. If the time values sampled at the two points are the same, which happens if the elapsed time between the two points is canceled by a backward time adjustment, then the speed computation requires division by zero. Adjusting clock rate, rather than clock value, still introduces the possibility of error. The error, however, is minimal since a small difference between the clock rate and the proper rate does not allow much error to accumulate during a short period of time.

[2] There is at least one historical instance in which both techniques were used. Pope Gregory XIII was concerned that the Julian calendar, which had a leap year every four years, was not keeping pace with the Spring equinox needed to compute the date for Easter. In 325, the year of the Council of Nicea, the Spring equinox was on 21 March. By 1582, it had slipped to 11 March. The pope effectively "moved the hands" of the clock (the calendar) by declaring that the day after 4 October 1582 would be designated 15 October 1582. He effectively "adjusted the spring" of the clock by declaring that years divisible by 100 but not by 400 would no longer be leap years. See Appendix A to the translation of Ptolemy's "The Almagest" in *Great Books of the Western World* Encyclopedia Britannica, Chicago, 1952, v. 16 p. 467.

Finally, adjusting clock rate, when properly implemented, minimizes the need to continually readjust the clock in the future. When the clock is running too fast or too slow, any adjustment of the clock value is only temporary. Since the clock is running at the wrong rate, the clock value, although initially correct, drifts from the proper value as time progresses. The clock continually needs readjustment. Only by correcting the clock rate itself can this drift be minimized.

2.2 ADJUSTABLE RATE CLOCK IMPLEMENTATION

Processors generally use crystal oscillators to drive their local realtime clocks. Crystal oscillator frequency is determined by factors such as manufacturing tolerance, temperature [3], and age. From the circuit design point of view, crystal frequency is considered fixed at a value within some tolerance of the desired value. The rest of the circuitry cannot directly adjust the crystal frequency. The problem is to implement a variable-rate local clock given that the frequency of the oscillator employed is not adjustable.

The solution to this problem can be understood by first considering a simpler one. Sometimes circuit designers need to implement a circuit that converts a signal oscillating at a high frequency into one oscillating at a lower frequency. For example, the designer of a processor board may want to use the same crystal, since crystals are expensive, to drive both a 30-MHz microprocessor integrated circuit and a 10-MHz local clock circuit. When one of the desired frequencies is a simple multiple of the other, the problem is easily solved using a **frequency divider** circuit. In this case, a divide-by-three frequency divider is needed since 10 MHz is one-third of 30 MHz. This frequency divider consists of nothing more than a counter circuit that outputs a pulse for every third pulse it receives as input. As shown in figure 1, the crystal is connected to the microprocessor integrated circuit and to the input of the frequency divider. The output of the frequency divider is connected to the local clock circuit. The 10-MHz input appears on the right side of the local clock block to emphasize the position of the local clock least-significant bit.

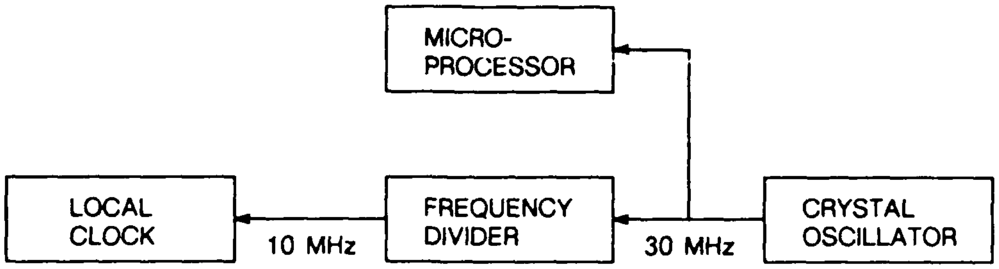

Figure 1. Frequency divider application example.

In the frequency divider example above, the frequency divider counter sequences through three states. By enumerating the states as 0, 1, and 2, the state sequence can be defined as a cycle of the form 0-1-2-0-1-2-... Whenever the counter reaches a particular state, for example state 0, it outputs a pulse to increment the local clock circuit. Thus the frequency divider counter increments at a rate three times faster than the local clock circuit.

[3] For a discussion of quartz crystal temperature coefficient of frequency, see Radio Society of Great Britain; *The Radio Communication Handbook*; 1968, p. 6.2.

The local clock oscillator input, obtained from the output of the frequency divider counter, cycles at the same rate that the frequency divider counter sequences once through all three states. Therefore, the states of the frequency divider counter represent the *phase* of the local clock oscillator cycle. For this reason, the frequency divider counter is called, hereafter, the **phase counter**. The local clock circuit, as defined by this example, is also a counter. Since the scope of the term "local clock circuit" is rather vague, this latter counter is called, hereafter, the **time counter**. As shown in figure 2, the crystal oscillator drives the phase counter, and the phase counter, in turn, drives the time counter. The time counter is the source of the time value seen by the application programmer.

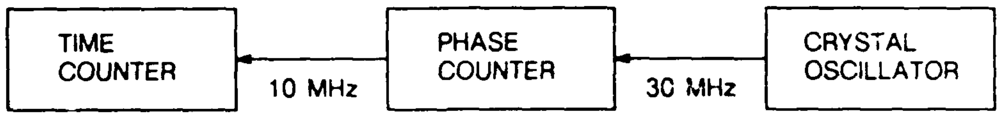

Figure 2. Local clock basic components.

Two components determine the frequency at which the time counter increments, the frequency of the oscillator and the number of sequential states in a single cycle of the phase counter. It is assumed that the frequency of the oscillator cannot be adjusted. Adjusting the rate at which the time counter increments, therefore, requires adjusting the number of sequential states in a single cycle of the phase counter.

Clock synchronization requires small changes in the frequency at which the time counter increments. This can be accomplished using the following two rules.

- If the time counter is too *slow*, periodically *delete* one of the states from the sequence of states executed by *one* of the cycles of the phase counter.
- If the time counter is too *fast*, periodically *insert* an additional state into the sequence of states executed by *one* of the cycles of the phase counter.

Deleting a phase counter state causes the respective phase cycle to complete more quickly. This, in turn, causes the time counter to increment faster for that cycle. Likewise, inserting a phase counter state causes the respective phase cycle to complete more slowly. This, in turn, causes the time clock to increment slower for that cycle.

Figure 3 shows an example of inserting a phase state in one cycle of a phase counter to slow the clock rate. Note that the vast majority of phase counter cycles are not modified. In the example, they have the cycle state sequence 0-1-2. Only occasionally is a cycle executed which is either shorter or longer than normal. The shorter and longer cycles have the cycle state sequences of 0-1 and 0-1-2-3, respectively.

The rate adjustment is controlled by controlling the frequency at which these modified phase cycles are introduced. The period between initiation of modified phase cycles is called the **rate adjustment period**. Figure 4 graphically illustrates how the inclusion of a modified phase state every rate adjustment period corrects the clock rate.

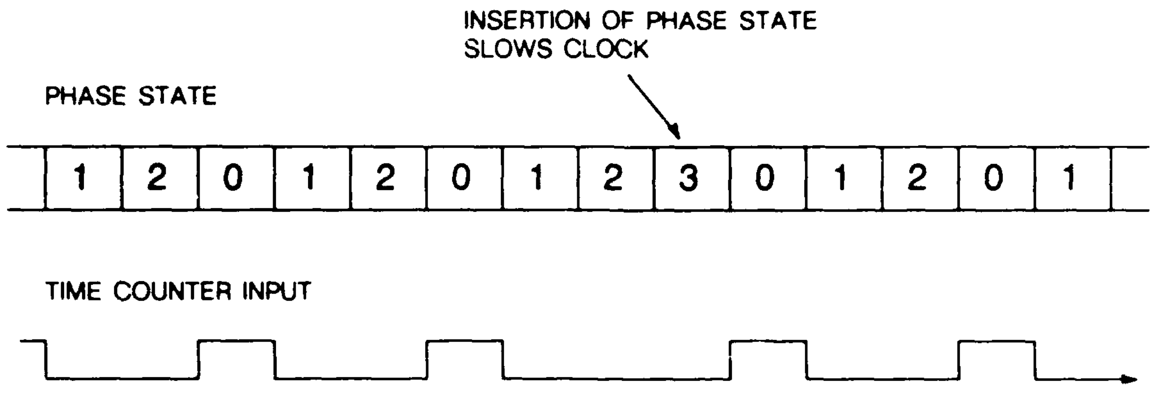

Figure 3. Phase state insertion.

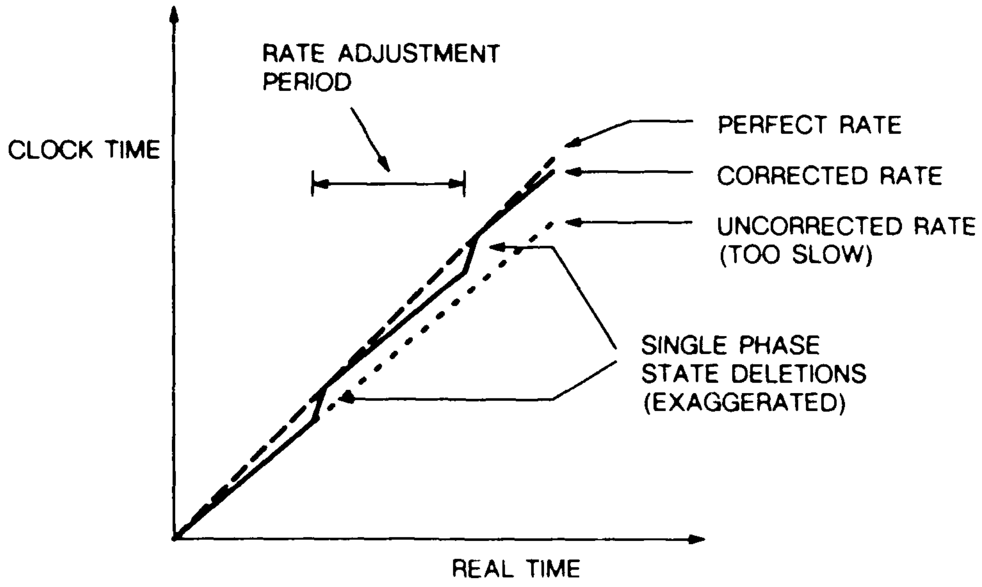
RATE ADJUSTMENT PERIOD
CLOCK TIME
PERFECT RATE
CORRECTED RATE
UNCORRECTED RATE (TOO SLOW)
SINGLE PHASE STATE DELETIONS (EXAGGERATED)
REAL TIME

Figure 4. Rate adjustment period.

The periodic introduction of modified cycles can be facilitated in hardware by inclusion of a third counter called the **rate adjustment counter**. The rate adjustment counter is loaded with an initial value selected by the software responsible for determining the magnitude of the local clock rate adjustment. This software is presented in section 4.3. The rate adjustment counter counts down, and upon reaching zero, sends a signal to the phase counter directing it to execute a single modified phase cycle rather than a normal phase cycle. The software also selects whether modified phase cycles are longer or shorter than normal. The rate adjustment counter then reloads the initial value and repeats the cycle. The reload maintains the current clock rate while awaiting the next adjustment from software.

Figure 5 is a block diagram showing how the three counters comprising the adjustable rate clock are interconnected to each other and to the internal data path to the processor. The module internal data bus provides the link between software and the local clock. The long select signal indicates that a long, rather than a short, phase sequence should be executed when a modified sequence is requested by the modify select signal.

Figure 6 shows a logic implementation for a divide-by-three phase counter employing these signals. The upper and lower flip-flops in the figure implement the most-significant and least-significant bits of the phase state respectively. The time counter clock signal is held high, and is thus inhibited, for all phase states except state 0.

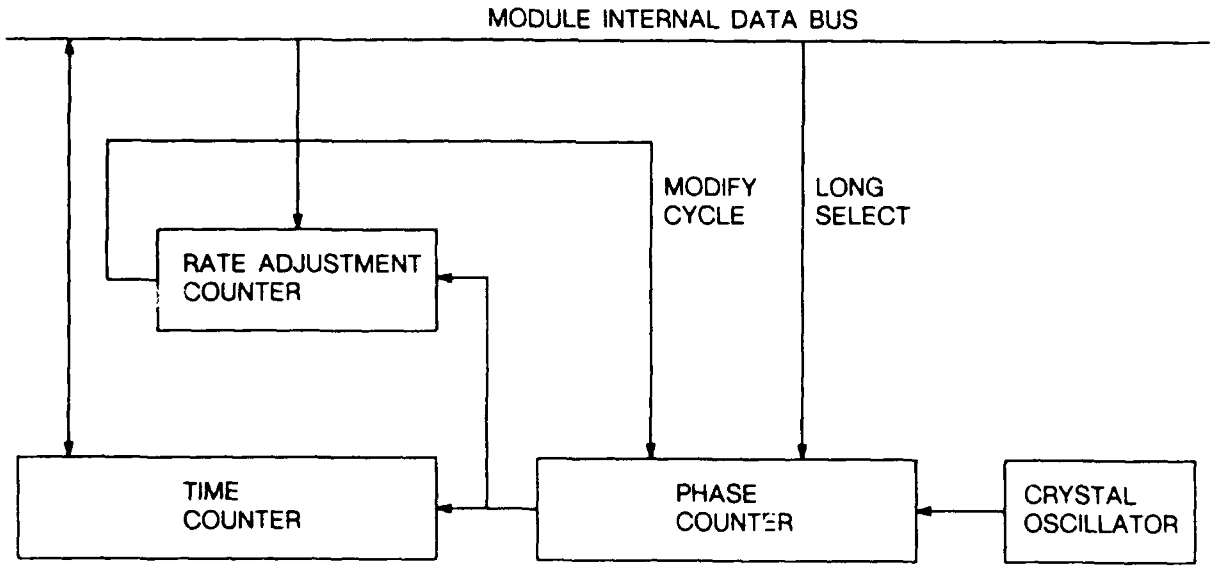

Figure 5. Adjustable rate clock block diagram.

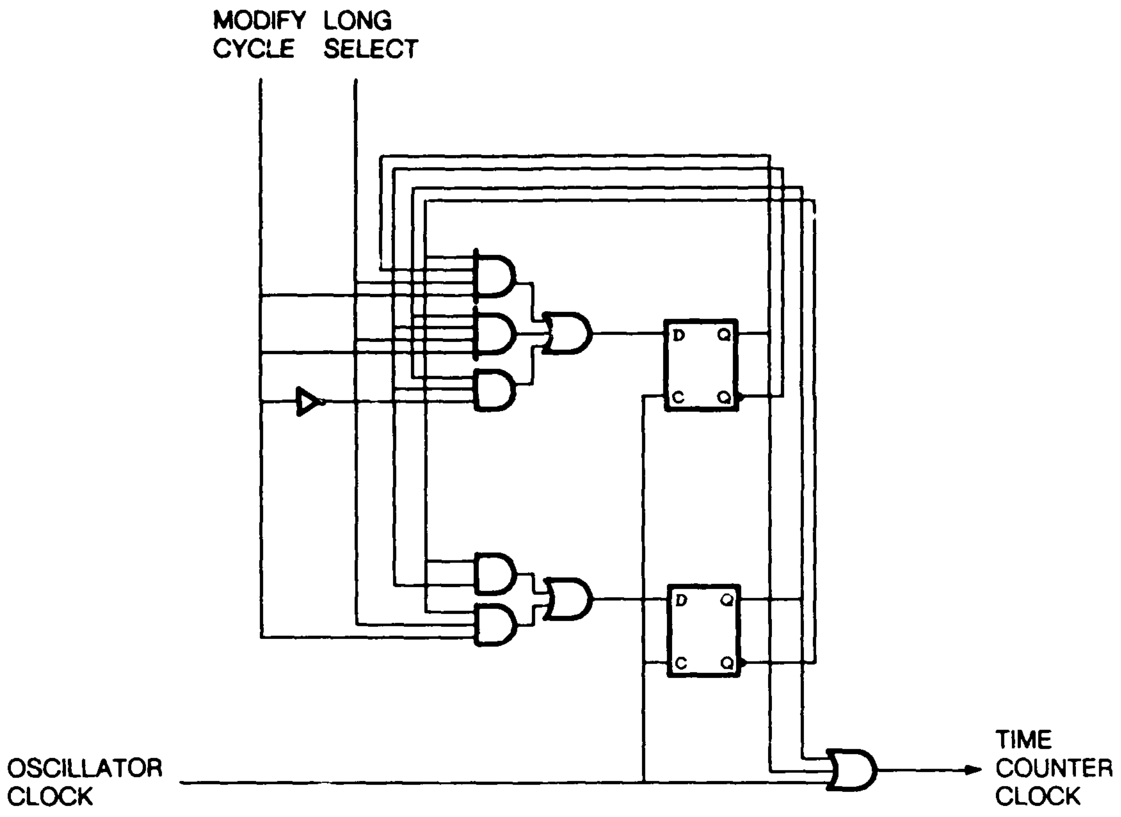

Figure 6. Divide-by-three phase counter implementation example.

3.0 RATE-MONOTONIC HARD-DEADLINE SCHEDULING

This section summarizes some of the fundamentals of the rate-monotonic hard-deadline realtime scheduling technology. The presentation is only intended as a quick introduction for those unfamiliar with the technology. For a more detailed discussion, including the derivations for the various equations, please see the papers referenced in the bibliography.

Rate monotonic scheduling is a means of scheduling the time allocated to periodic hard-deadline realtime users of a resource. The users are assigned priorities such that a shorter fixed period between deadlines is associated with a higher priority. Rate monotonic scheduling provides a low-overhead reasonably resource-efficient means of guaranteeing that all users will meet their deadlines provided that certain analytical equations are satisfied during the system designed. It avoids the design complexity of time-line scheduling and the overhead of dynamic approaches such as earliest-deadline scheduling.

3.1 TASK SCHEDULING

Rate monotonic scheduling was first conceived in the context of scheduling periodic hard-deadline realtime tasks on a processor. The seminal paper by Liu and Layland [1973] constrained the tasks as follows:

- Task requests are periodic, with a constant interval between requests.
- Each task must be completed before its next request occurs.
- Task requests are independent; no task request depends on another task state.
- Uninterrupted execution time of each task does not exceed a known constant.

The processor **utilization factor** is defined as the "fraction of processor time spent in executing the task set." Letting T and C represent the period between task requests and the task uninterrupted execution time, respectively, the utilization factor U for a processor executing N tasks is computed by the equation

$$U = \sum_{i=1}^{N} \frac{C_i}{T_i} \quad .$$

If the tasks are assigned priorities such that for each pair of tasks the one with the shorter period has a higher priority, then all tasks are guaranteed to meet their deadlines provided that the following equation is true:

$$U \leq N(2^{1/N} - 1) \quad .$$

In the *worst case*, the right side of this equation, called the **schedulability bound**, approaches *ln* 2 (69.3 percent) as N approaches infinity. Typical applications are around 90 percent.

It has been shown [Lehoczky, Sha, and Ding, 1987] that the Liu and Layland utilization bound can be exceeded provided that the following equation is satisfied:

$$\forall\, i,\ 1 \le i \le N$$

$$\min_{(k,l) \in R_i} \left[\sum_{j=1}^{i} C_j \frac{1}{l\, T_k} \operatorname{ceil}\left(\frac{l\, T_k}{T_j}\right)\right] \le 1$$

$$R_i = \{ j, k \mid 1 \le k \le i,\ l = 1, \ldots, \operatorname{floor}(T_i/T_k)\}.$$

Often applications have periodic tasks whose uninterrupted execution time is not a constant. When the variation in execution time is slight, one can use the maximum execution time without much loss. But when the maximum execution time is large compared to the average and occurs relatively infrequently, it may be beneficial to allow some tasks to potentially miss their deadlines in order to maintain high processor utilization. The tasks missing their deadlines are the ones with the longest periods since they are assigned the lowest priority. Unfortunately, they may be the most critical to the application. The problem can be solved by partitioning each critical task with a long period into several tasks with shorter periods. This technique is called **period transformation** [Sha, Lehoczky, and Rajkumar, 1986]. Elimination of a long period in the task set may also improve the schedulability bound.

Rate monotonic scheduling has been extended to include aperiodic tasks. Aperiodic tasks are characterized by response-time deadlines rather than periodic deadlines. They preempt periodic tasks as long as so doing does not cause a periodic task to miss its deadline. This is accomplished by implementing a server task to execute aperiodic tasks. The server task executes whenever there are *both* aperiodic tasks pending *and* the server has not exhausted its designed maximum execution time within its designed period. It has been shown [Lehoczky, Sha, and Strosnider, 1987] that all periodic deadlines are guaranteed to be met provided that *both* the total system utilization factor satisfies the equation

$$U \le U_a + \ln \frac{2 + U_a}{2\, U_a + 1}$$

$$U_a = \frac{C_a}{T_a} = \frac{\textit{server maximum execution time}}{\textit{server period}},$$

and the period of the server is less than the period of the next-highest priority task minus the maximum execution time of the server.

3.2 TASK COORDINATION

The rate monotonic scheduling technology has also been extended to account for tasks that are dependent due to requirements for exclusive access to shared resources. Traditional semaphores are inadequate because of **priority inversion**. Priority inversion occurs whenever a high-priority task is prevented from entering a critical region because a low-priority task gained access to the critical region first. The high-priority task must wait not only for the low-priority task to execute the critical region, but also for any additional time that the low-priority task is delayed because it is preempted by other tasks having medium priority as illustrated in figure 7. The locked critical region has effectively dragged the priority of the high-priority task down to a level below that of the low-priority task, violating the task priority requirements needed to guarantee that deadlines are met.

The priority inversion problem is reduced by minimizing the uninterrupted execution time of critical regions and by implementing **priority inheritance**. Priority inheritance means that when a task blocks higher-priority tasks from entering a critical region, the task *inherits* the priority of the highest-priority task it is blocking long enough to exit the critical region. Temporarily armed with this higher priority, the task

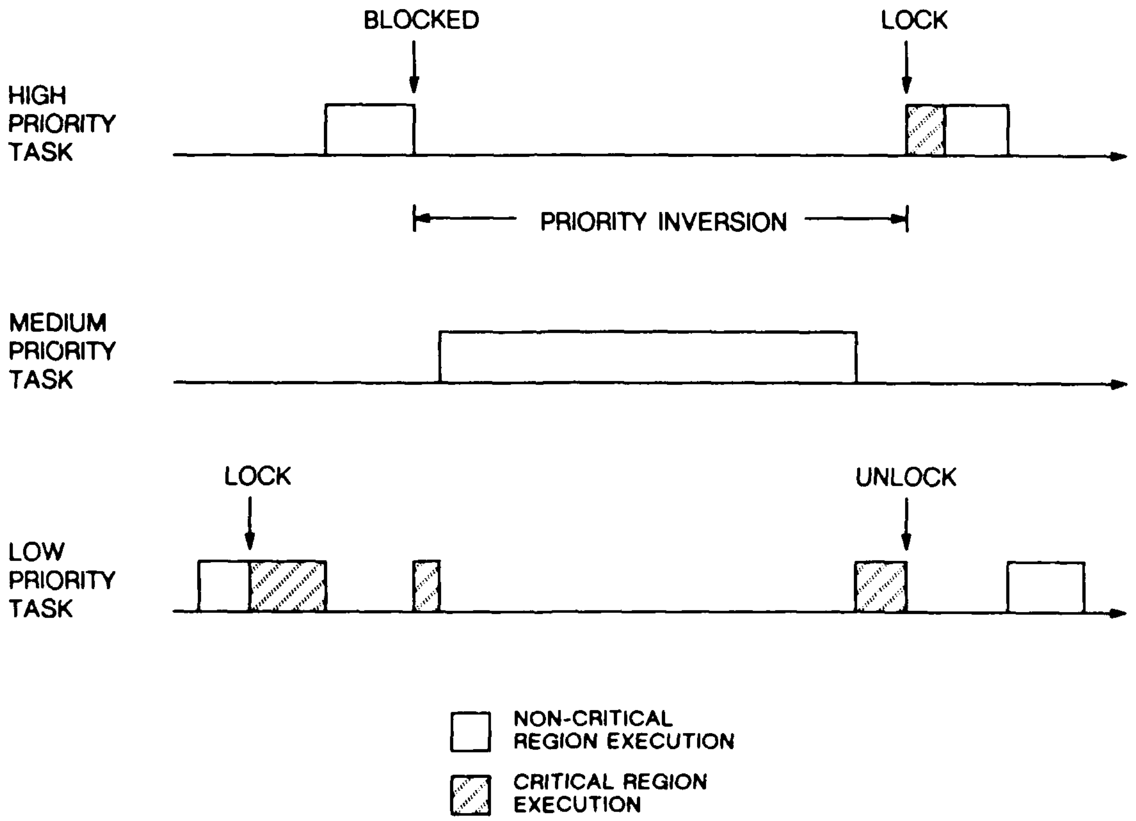

Figure 7. Priority inversion example.

avoids preemption by medium-priority tasks whose priority is lower than the inherited priority. Priority inversion is still present while the task is blocking the higher-priority tasks, but it is now no greater than the uninterrupted execution time of the critical region.

Deadlocks are avoided by combining the priority inheritance concept with the **priority ceiling** concept [Goodenough and Sha, 1988]. The priority ceiling of a semaphore is the priority of the highest-priority task that could ever lock the semaphore. A task is permitted to lock a semaphore only if the priority of the task is greater than the priority ceilings of all the semaphores already locked by other tasks. When a task is not permitted to lock the semaphore, the task is suspended. Thus semaphores are only locked in the sequence of increasing priority ceiling. Since they cannot be locked in a loop sequence, there is no deadlock.

Letting B represent the worst-case delay due to blocking experienced by a task, the utilization factor of a system implementing priority inheritance and priority ceiling is given the by the equation

$$U = \sum_{i=1}^{N} \frac{C_i}{T_i} + \max_{i=1}^{N} \left(\frac{B_i}{T_i} \right) .$$

The utilization bound can be exceeded provided that the following equation is satisfied:

$$\forall\, i,\ 1 \le i \le N$$

$$\min_{(k,l) \in R_i} \left[\sum_{j=1}^{i-1} C_j \frac{1}{l\,T_k} \operatorname{ceil}\left(\frac{l\,T_k}{T_j} \right) + \frac{C_i}{l\,T_k} + \frac{B_i}{l\,T_k} \right] \le 1$$

$$R_i = \{\, j,k \mid 1 \le k \le i,\ l = 1, \ldots, \operatorname{floor}(T_i/T_k) \} .$$

4.0 CLOCK SYNCHRONIZATION

The basic approach to clock synchronization can be summarized as follows. Periodically, the clock **synchronization algorithm** is executed. It is executed frequently enough that the clocks never drift beyond a margin of synchronization error selected by the system designer. For each iteration of the algorithm, one of the modules simultaneously broadcasts to all modules, including itself, a command requesting that each module immediately sample and locally store the value of its respective clock. When a module detects that a sample has been taken, the module is responsible for providing whatever adjustment is necessary to bring its own clock into synchronization. The method by which it synchronizes its own clock is called the **adjustment algorithm.** There are many variations of adjustment algorithms. All of them, however, make their adjustment based on comparison of their own clock sample against clocks on other modules.

This section describes the operational details and motivation behind this clock synchronization approach. It also shows how the approach can be incorporated within the rate-monotonic hard-deadline realtime scheduling technology.

4.1 SYNCHRONIZATION ALGORITHM EXECUTION FREQUENCY

The **maximum permissible elapsed time** between consecutive invocations of the synchronization algorithm is easily determined from the maximum permissible synchronization error and the accuracy of the clock oscillator. For example, if the maximum permissible synchronization error is 500 ns and the accuracy of the oscillator is ±0.01 percent, the maximum permissible time between invocations is 500 ns / 0.0001 = 5 ms. The system designer can lengthen the maximum permissible elapsed time between invocations by either relaxing the permissible synchronization error or by improving the accuracy of the clock oscillator [4].

The synchronization algorithm can be executed as a periodic task within the context of the rate monotonic scheduling technology. During each iteration, the task is blocked until a new set of simultaneous clock samples are available. This blocking does not create a priority inversion problem since the clock value remains available to other tasks.

The maximum permissible elapsed time between consecutive invocations of the synchronization algorithm should not be confused with the period of the periodic task implementing it. Rate monotonic scheduling (with a suitable utilization factor) guarantees that the periodic task will always meet its deadline. It does not, however, force a particular proximity between the time the task supplies a new clock adjustment and the deadline for such an adjustment. Depending upon the priorities of the other ready tasks, it may be executed far in advance of the deadline or just in time for the deadline. It is possible for the task to adjust the clock very early in one period and very late in the next period and still meet both respective deadlines. Thus, the worst case elapsed time

[4] It is also possible to lengthen the maximum permissible elapse time given knowledge of the maximum rate of oscillator *frequency change*, that is, the maximum acceleration in clock value.

between adjustments is twice the period [5] between deadlines minus the unblocked adjustment execution time. This is illustrated by figure 8. Usually the latter execution time is negligible by comparison to the period. In the example above, the maximum permissible elapsed time between adjustments was 5 ms. In order to guarantee this limit, the periodic task under rate monotonic scheduling needs a period of half that figure, or 2.5 ms.

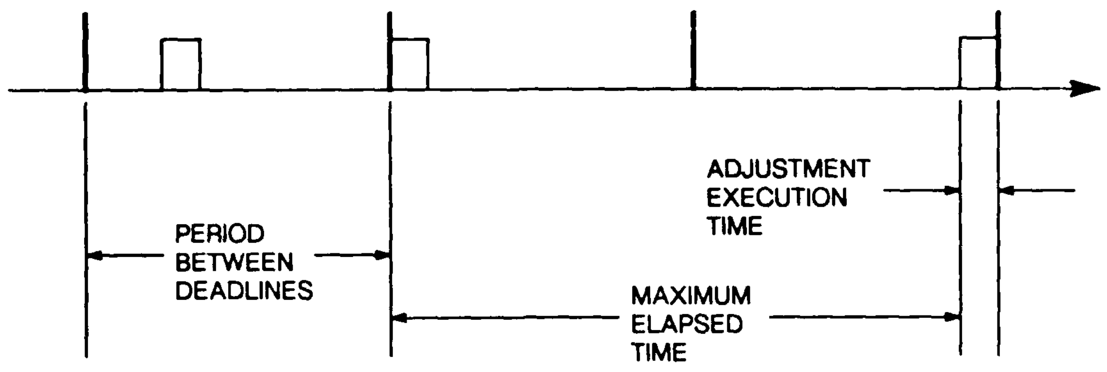

Figure 8. Periodic task period computation.

4.2 SIMULTANEOUS SAMPLING OF CLOCKS

The simultaneous sampling of the clocks to be synchronized, the ability to obtain an instantaneous snapshot of their values, is a key component to the periodic phase adjustment approach. It is not important precisely when the local clocks are sampled as long as the maximum permissible elapsed time between samples is not exceeded. It is important that they be sampled simultaneously.

Many alternative clock synchronization approaches require that their associated tasks and messages execute at the highest priority. This is not the case with the phase adjustment approach presented here. Since there is no requirement that a sample be taken at a precise time, only that it be taken frequently enough and simultaneously, there is no problem created by delays due to contention at the processor or on the access path to the various clocks.

Simultaneous sampling usually requires some hardware support. Three applications are considered below. The first is synchronization over a high-performance parallel backplane bus supporting broadcast where bus propagation delay is not considered an issue. The second is synchronization through an interface, bus repeater, or gateway through which instantaneous broadcast is not always possible. The third is synchronization over a token-passing local area network where bus propagation delay is significant in comparison to the synchronization accuracy desired.

[5] The two-to-one relationship suggests affinity to the Nyquist sampling principle. The Nyquist sampling principle specifies the minimum frequency at which one must sample to avoid losing signal information. Since the Nyquist rate is a constant for a given signal, the maximum period between samples is also constant. The problem of variations in task execution time, described above, is also one of limiting the maximum period between samples. In the latter case, the information contained in the signal is the frequency of the uncorrected clock synchronization phase error.

When the local clocks are located on separate processor modules (boards) interconnected through one or more high-performance parallel backplane buses, the clocks can be sampled simultaneously by having one of the processors broadcast a **sample command** over one of the backplane buses. Since the command is a broadcast and bus propagation delay is not considered an issue, all the processor modules see the sample command simultaneously.

As mentioned previously, all sample commands are normally generated by the same module. To avoid creating a potential single point of failure, modules are equipped with both the ability to generate sample commands and to monitor whether sample commands are, in fact, being generated. Each such module has a **watchdog timer** whose interval is uniquely determined by the module geographical address. A module initializes its watchdog timer when it generates a sample command. It also initializes its watchdog timer when it detects that a sample command has been generated by another module. A module generates a sample command when its watchdog timer interval expires. Normally, the module with the shortest watchdog timer interval generates all sample commands. This is because all watchdog timers are initialized at the same time and the ones on the other modules, whose initialized intervals are longer, do not have the opportunity to expire before being reinitialized. In the event that the module expected to generate sample commands fails, the module with the next successively-longer watchdog timer interval automatically takes responsibility for continuing to generate sample commands.

Sample commands access a bus address dedicated to the clock sample function. The access only touches the address; no time data is read or written [6]. When the hardware on each processor board detects access to that address, it generates a signal to atomicly load the current value of its local clock into a **sample register**. The same signal can also generate an interrupt to the processor to indicate that new clock samples are available. This signal permits the adjustment algorithm to proceed.

When clock synchronization is required across an interface, it is often the case that a broadcast sample command on one side of the interface cannot be instantaneously presented by the interface to the other side. The problem is further complicated when the other side connects to a shared resource where variable delays due to contention are present, such as connections to a multiplexed-peripheral or a bus gateway. The problem can be solved by providing the interface with an internal timer that measures the elapsed time the sample command uses in crossing the interface. The elapsed time is then used as a correction factor for interpreting time values obtained across the interface.

When the local clocks are situated on separate nodes interconnected through a token-passing local area network, the sample command can be efficiently distributed by dedicating the highest-priority token for that purpose [7]. The token has the property that all nodes "see" the token as it passes, yet none of them "claim" the token except the node that created it (so the node can remove it after all nodes have seen it). Upon seeing the token, each processor samples its local clock and generates an interrupt to its processor as was the case for the backplane bus.

---

[6] Sample commands merely signify, at the rather arbitrary moment of their broadcast, a time at which all clocks must be simultaneously sampled. They provide a coordination function, not a data function. They can be implemented on the IEEE 896 Futurebus as an "address beat."

[7] This approach is the inspiration of Dan Green of the Naval Surface Weapons Center.

Since the token has the highest priority and no node claims the token for network traffic, the token passes quickly around the local area network. The token thus approximates a broadcast. The approximation is not perfect, however, due to propagation delays in the network. The error in each time sample is the propagation delay between the node that generated the time sample token and the node that took the particular sample in question. Since the token passes from node to node without ever being claimed or preempted by a token of higher priority, the propagation delay for the token to pass from any node to any other node is a constant for the respective pair of nodes. The time samples are corrected by subtracting the appropriate propagation delay constants from the time sample values.

The use of the highest-priority token may appear to contradict the statement that messages employed by the phase adjustment approach do not require the highest priority. The statement is still correct when one realizes that the highest-priority token is used solely to preserve the broadcast character of a single message *during* transit. The initial release of the token may be at any appropriate priority. But once it is released, it must circulate unimpeded so that transit delay times between nodes are known constants.

4.3 RATE ADJUSTMENT ALGORITHMS

The rate adjustment algorithm is implemented as an independent task on each module participating in clock synchronization. It is responsible for estimating the synchronization phase error of the local clock based upon sample values from other clocks, and for making adjustments to the local clock rate in order to gradually eliminate this synchronization phase error.

On some systems, all the local clocks are of equal quality. In other words, the sample value obtained from one clock is considered no more accurate or reliable than that obtained from another. For these systems, the synchronization phase error can be estimated by taking the simple average of the synchronization phase differences between the clock on the module performing the algorithm and each of the respective clocks on the other modules.

On other systems, a module containing an interface to a highly-accurate external clock, such as a cesium standard, is considered the standard of true time to which all other clocks must be synchronized. For these systems the synchronization phase error of the local clock is simply the difference between its sample value and the sample value at the external clock interface module.

Finally, between these two extremes, there are systems in which no single clock is considered as perfect and in which some clocks are better than others. For these systems, the synchronization phase error is estimated using a weighted average where the coefficients reflect the quality attributed to the sample from each clock.

A weighted average can be used to describe the computation for all three types of systems. In the case where all the clocks were of equal quality, the coefficients of the average are identical. In the case where only one clock was considered the source of true time, all but one of the coefficients are zero.

Before actually performing the average, it is beneficial to check the various sample values for any that appear wildly out of range. Wide variations can indicate clocks or access paths that have failed in some way. They can also indicate clocks that were improperly initialized. Sample values from these clocks should be either excluded completely, or possibly included at greatly reduced weight.

Once the local clock synchronization phase error has been estimated, all that remains is to adjust the local clock rate. How quickly one causes the elimination of the synchronization phase error over time is largely left to the system designer. It is important, however, not to cause such a drastic change in rate that the gradual correction in synchronization phase error overshoots and becomes unstable. This can be avoided by selecting an adjustment that corrects the error no faster than half the maximum permissible elapsed time between samples.

5.0 CONCLUSIONS

A distributed set of local clocks associated with the respective processors of a distributed processing system generally has the advantages of faster access, greater accuracy, and greater fault tolerance than obtainable from a centralized clock.

Local clock adjustment implemented by modifying the rate, rather than the value, of the clock has the advantages of preventing backward adjustment in time value, avoiding abrupt discontinuities in the steady advance of time value, and minimizing the need for future adjustments. A variable-rate clock derived from a fixed-rate oscillator can be implemented by inserting a programmable frequency-divider circuit, called a phase counter, between the oscillator and the time counter maintaining the user-accessible time value.

The periodic phase adjustment approach to clock synchronization adjusts a local clock based upon an average of simultaneously-sampled time values collected from various clocks. Use of the broadcast simultaneous-sample command, rather than synchronized transmission of time value, has the advantage that clock synchronization processing and message traffic do not require the highest priority in order to ensure accuracy. The approach is compatible with rate monotonic scheduling. The approach supports both systems with and without external time standards. The approach can be extended to handle non-broadcast interfaces and local area networks where propagation delay is a factor.

6.0 REFERENCES

Goodenough, J. B. and L. Sha. 1988. "The Priority Ceiling Protocol: A Method of Minimizing the Blocking of High Priority Ada Tasks," *Ada Letters*, vol. 8, no. 7, pp. 20-31.

Jefferson, D. R. July 1985. "Virtual TIme," *ACM Trans. Prog. Languages*, vol. 7, no. 3, pp. 404-425.

Lamport, L. April 1984. "Using Time Instead of Timeout for Fault-Tolerant Distributed Systems," *ACM Trans. Prog. Languages*, vol. 6, no. 2, pp. 254-280.

Lehoczky, J. P., L. Sha, and Y. Ding. 1987. *The Rate Monotonic Scheduling Algorithm — Exact Characterization and Average Case Behavior*. Tech. Rep., Dept. Statistics, Carnegie-Mellon University.

Lehoczky, J. P., L. Sha, and J. K. Strosnider. 1987. "Enhanced Aperiodic Responsiveness in Hard Real-Time Environments," *Proceedings Real-Time Systems Symposium, IEEE,* pp. 261-270.

Liu, C. L. and J. W. Layland. January 1973. "Scheduling Algorithms for Multiprogramming a Hard Real-Time Environment," *Journal ACM*, vol. 20, no. 1, pp. 46-61.

Sha, L., J. P. Lehoczky, and R. Rajkumar. 1986. "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling," *Proceedings Real-Time Systems Symposium, IEEE*, pp. 181-191.

7.0 BIBLIOGRAPHY

7.1 SYNCHRONIZATION AND USE OF DISTRIBUTED REALTIME CLOCKS

Cole, R. and C. Forcroft. December 1988. "An Experiment in Clock Synchronization"; *Computer Journal,* vol. 31, no. 6, pp. 496-502.

Dolev, D., J. Y. Halpern, and R. H. Strong. April 1986. "On the Possibility and Impossibility of Achieving Clock Synchronization"; *Journal of Computer and System Science,* vol. 32, no. 2, pp. 230-250.

Halpern, J. Y., B. Simons, R. H. Strong, and D. Dolev. August 1984. "Fault-Tolerant Clock Synchronization"; *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing,* ACM, Vancouver, B.C., Canada, pp. 89-102.

Jefferson, D. R. July 1985. "Virtual Time"; *ACM Trans. Prog. Languages,* vol. 7, no. 3, pp. 404-425.

Kopetz, H. and W. Ochsenreiter. August 1987. "Clock Synchronization in Distributed Real-Time Systems"; *IEEE Trans. Computers,* vol. C-36, no. 8, pp. 933-940.

Kopetz, H., A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. February 1989. "Distributed Fault-Tolerant Real-Time Systems: The Mars Approach"; *IEEE Micro,* vol. 9, no. 1, pp. 25-40 (clock sync. pp. 31-32).

Lamport, L. and P. M. Melliar-Smith. August 1984. "Byzantine Clock Synchronization"; *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing,* ACM, Vancouver, B.C., Canada, pp. 68-74.

Lamport, L. and P. M. Melliar-Smith. January 1985. "Synchronizing Clocks in the Presence of Faults"; *Journal ACM,* vol. 32, no. 1, pp. 52-78.

Lamport, L. April 1984. "Using Time Instead of Timeout for Fault-Tolerant Distributed Systems"; *ACM Trans. Prog. Languages,* vol. 6, no. 2, pp. 254-280.

Lundelius, J. and N. Lynch. August 1984. "A New Fault-Tolerant Algorithm for Clock Synchronization"; *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing,* ACM, Vancouver, B.C., Canada, pp. 75-88.

Molle, M. L. and L. Kleinrock. September 1985. "Virtual Time CSMA: Why Two Clocks Are Better than One"; *IEEE Trans. Communications,* vol. COM-33, no. 9, pp. 919-933.

Srikanth, T. K. and S. Toueg. July 1987. "Optimal Clock Synchronization"; *Journal ACM,* vol. 34, no. 3, pp. 626-645.

Volz, R. and T. N. Mudge. August 1987. "Instruction Level Timing Mechanisms for Accurate Real-Time Task Scheduling"; *IEEE Trans. Computers,* vol. C-36, no. 8, pp. 988-993.

Volz, R. A.; and T. N. Mudge. April 1987. "Timing Issues in the Distributed Execution of Ada Programs"; *IEEE Trans. Computers,* vol. C-36, no. 4, pp. 449-459.

7.0 BIBLIOGRAPHY (Cont'd)

7.2 RATE MONOTONIC HARD-DEADLINE REALTIME SCHEDULING TECHNOLOGY

Cornhill, D. L. and L. Sha. November-December 1987. "Priority Inversion in Ada or What Should Be the Priority of an Ada Server Task?"; *Ada Letters,* vol. 7, no. 7, pp. 30-32.

Goodenough, J. B. and L. Sha. Fall 1988. "The Priority Ceiling Protocol: A Method of Minimizing the Blocking of High Priority Ada Tasks"; *Proceedings 2nd ACM International Workshop on Real-Time Ada Issues, Ada Letters,* vol. 8, no. 7, pp. 20-31.

Lehoczky, J. P., L. Sha, and Y. Ding. 1987. *The Rate Monotonic Scheduling Algorithm — Exact Characterization and Average Case Behavior,* Tech. Rep., Dept. Statistics, Carnegie-Mellon University.

Lehoczky, J. P., L. Sha, and J. K. Strosnider. 1987. "Enhanced Aperiodic Responsiveness in Hard Real-Time Environments"; *Proceedings Real-Time System Symposium,* IEEE, pp. 261-270.

Levine, G. Fall 1988. "The Control of Priority Inversion in Ada"; *Proceedings 2nd ACM International Workshop on Real-Time Ada Issues, Ada Letters,* vol. 8, no. 7, pp. 53-56.

Liu, C. L. and J. W. Layland. January 1973. "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment"; *Journal ACM,* vol. 20, no. 1, pp. 46-61.

Sha, L., J. P. Lehoczky, and R. Rajkumar. December 1986. "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling"; *Proceedings Real-Time System Symposium,* IEEE, New Orleans, Louisiana, pp. 181-191.

Sha, L., R. Rajkumar and J. P. Lehoczky. 1987. *Priority Inheritance Protocols: An Approach to Real-Time Synchronization,* Tech. Rep., Dept. Computer Science, Carnegie-Mellon University, to appear in IEEE Trans. on Computers.

Sprunt, B., D. Kirk, and L. Sha. June 1988. "Priority-Driven Preemptive I/O Controllers for Real-Time Systems"; *Proceedings 15th Annual International Symposium on Computer Architecture,* IEEE.

Strosnider, J. K., T. Marchok, and J. Lehoczky. 1988. "Advanced Real-Time Scheduling Using the IEEE 802.5 Token Ring"; *Proceedings Real-Time System Symposium,* IEEE, pp. 42-52.